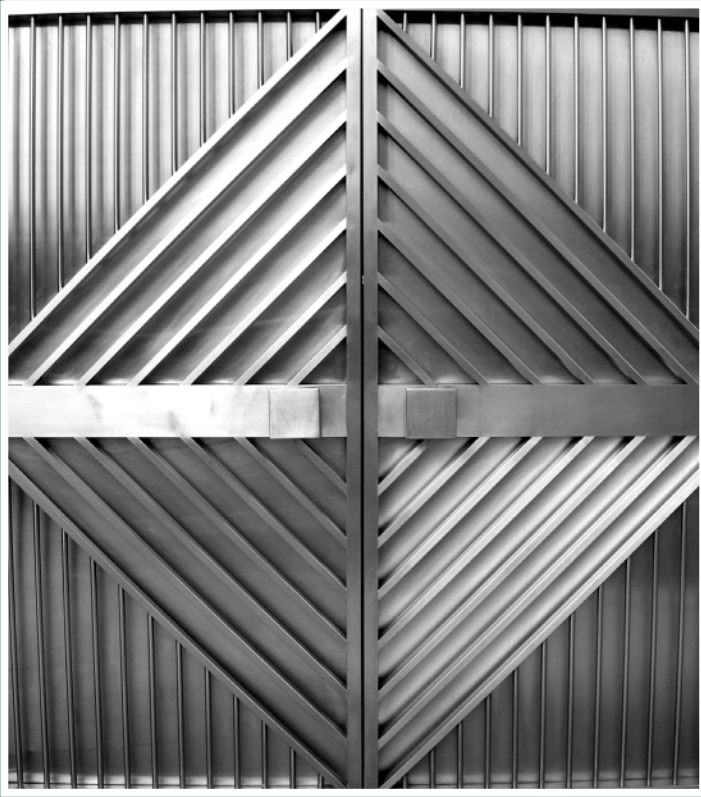


Chapter 10



Service API and Contract Versioning with Web Services and REST Services

- 10.1 Versioning Basics
- 10.2 Versioning and Compatibility
- 10.3 REST Service Compatibility Considerations
- 10.4 Version Identifiers
- 10.5 Versioning Strategies
- 10.6 REST Service Versioning Considerations

NOTE

This chapter provides a number of code examples that help demonstrate various versioning scenarios and approaches. Note that these code examples are not related to any code examples provided in Case Study Examples from preceding chapters.

After a service contract is deployed, consumer programs will naturally begin forming dependencies on it. When we are subsequently forced to make changes to the contract, we need to figure out:

- Whether the changes will negatively impact existing (and potentially future) service consumers
- How changes that will and will not impact consumers should be implemented and communicated

These issues result in the need for versioning. Anytime you introduce the concept of versioning into an SOA project, a number of questions will likely be raised, for example:

- What exactly constitutes a new version of a service contract? What's the difference between a major and minor version?
- What do the parts of a version number indicate?
- Will the new version of the contract still work with existing consumers that were designed for the old contract version?
- Will the current version of the contract work with new consumers that may have different data exchange requirements?
- What is the best way to add changes to existing contracts while minimizing the impact on consumers?
- Will we need to host old and new contracts at the same time? If yes, for how long?

We will address these questions and provide a set of options for solving common versioning problems. The upcoming sections begin by covering some basic concepts, terminology, and strategies specific to service contract versioning.

10.1 Versioning Basics

So when we say that we're creating a new version of a service contract, what exactly are we referring to? The following sections explain some fundamental terms and concepts and further distinguish between Web service contracts and REST service contracts.

Versioning Web Services

As we've established many times in this book, a Web service contract can be comprised of several individual documents and definitions that are linked and assembled together to form a complete technical interface.

For example, a given Web service contract can consist of:

- One (sometimes more) WSDL definitions
- One (usually more) XML Schema definitions
- Some (sometimes no) WS-Policy definitions

Furthermore, each of these definition documents can be shared by other Web service contracts. For example,

- A centralized XML Schema definition will commonly be used by multiple WSDL definitions.
- A centralized WS-Policy definition will commonly be applied to multiple WSDL definitions.
- An abstract WSDL description can be imported by multiple concrete WSDL descriptions or vice versa.

Of all the different parts of a Web service contract, the part that establishes the fundamental technical interface is the abstract description of the WSDL definition. This represents the core of a Web service contract and is then further extended and detailed through schema definitions, policy definitions, and one or more concrete WSDL descriptions.

When we need to create a new version of a Web service contract, we can therefore assume that there has been a change in the abstract WSDL description or one of the contract documents that relates to the abstract WSDL description. The Web service contract content commonly subject to change is the XML schema content that provides the types for the abstract description's message definitions. Finally, the one other contract-related

technology that can still impose versioning requirements but is less likely to do so simply because it is a less common part of Web service contracts is WS-Policy.

Versioning REST Services

If we follow the REST model of using a uniform contract to express service capabilities, the sharing of definition documents between service contracts is even clearer. For example,

- All HTTP methods used in contracts are standard across the architecture.
- XML Schema definitions are standard, as they are wrapped up in general media types.
- The identifier syntax for lightweight service endpoints (known as resources) are standard across the architecture.

Changes to the uniform contract facets that underlie each service contract can impact any REST service in the service inventory.

Fine and Coarse-Grained Constraints

Regardless of whether XML schemas are used with Web services or REST services, versioning changes are often tied to the increase or reduction of the quantity or granularity of constraints expressed in the schema definition. Therefore, let's briefly recap the meaning of the term *constraint granularity* in relation to a type definition.

Note the bolded and italicized parts in Example 10.1:

```
<xsd:element name="LineItem" type="LineItemType"/>
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
      namespace="##any" processContents="lax"/>
  </xsd:sequence>
  <xsd:anyAttribute namespace="##any"/>
</xsd:complexType>
```

Example 10.1

A `complexType` construct containing fine and coarse-grained constraints.

As indicated by the bolded text, there are elements with specific names and data types that represent parts of the message definition with a *fine* level of constraint granularity. All the message instances (the actual XML documents that will be created based on this structure) must conform to these constraints to be considered valid (which is why these are considered the absolute “minimum” constraints).

The italicized text shows the element and attribute wildcards also contained by this complex type. These represent parts of the message definition with an extremely *coarse* level of constraint granularity in that messages do not need to comply to these parts of the message definition at all.

The use of the terms “fine-grained” and “coarse-grained” is highly subjective. What may be a fine-grained constraint in one contract may not be in another. The point is to understand how these terms can be applied when comparing parts of a message definition or when comparing different message definitions with each other.

10.2 Versioning and Compatibility

The number one concern when developing and deploying a new version of a service contract is the impact it will have on other parts of the enterprise that have formed or will form dependencies on it. This measure of impact is directly related to how compatible the new contract version is with the old version and its surroundings in general.

This section establishes the fundamental types of compatibility that relate to the content and design of new contract versions and also tie into the goals and limitations of different versioning strategies introduced at the end of this chapter.

Backwards Compatibility

A new version of a service contract that continues to support consumer programs designed to work with the old version is considered *backwards compatible*. From a design perspective, this means that the new contract has not changed in such a way that it can impact existing consumer programs that are already using the contract.

Backwards Compatibility in Web Services

Example 10.2 provides a simple instance of a backwards-compatible change based on the addition of a new operation to an existing WSDL definition:

```

<definitions name="Purchase Order" targetNamespace=
  "http://actioncon.com/contract/po"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://actioncon.com/contract/po"
  xmlns:po="http://actioncon.com/schema/po">
  ...
  <portType name="ptPurchaseOrder">
    <operation name="opSubmitOrder">
      <input message="tns:msgSubmitOrderRequest" />
      <output message="tns:msgSubmitOrderResponse" />
    </operation>
    <operation name="opCheckOrderStatus">
      <input message="tns:msgCheckOrderRequest" />
      <output message="tns:msgCheckOrderResponse" />
    </operation>
    <operation name="opChangeOrder">
      <input message="tns:msgChangeOrderRequest" />
      <output message="tns:msgChangeOrderResponse" />
    </operation>
    <operation name="opCancelOrder">
      <input message="tns:msgCancelOrderRequest" />
      <output message="tns:msgCancelOrderResponse" />
    </operation>
    <operation name="opGetOrder">
      <input message="tns:msgGetOrderRequest" />
      <output message="tns:msgGetOrderResponse" />
    </operation>
  </portType>
</definitions>

```

Example 10.2

The addition of a new operation represents a common backwards-compatible change.

By adding a brand-new operation, we are creating a new version of the contract, but this change is backwards-compatible and will not impact any existing consumers. The new service implementation will continue to work with old service consumers because all the operations that an existing service consumer might invoke are still present and continue to meet the requirements of the previous service contract version.

Backwards Compatibility in REST Services

A backwards-compatible change to a REST-compliant service contract might involve adding some new resources or adding new capabilities to existing resources. In each of these cases the existing service consumers will only invoke the old methods on the old resources, which continue to work as they previously did.

As demonstrated in Example 10.3, supporting a new method that existing service consumers don't use results in a backwards-compatible change. However, in a service inventory with multiple REST services, we can take steps to ensure that new service consumers will continue to work with old versions of services.

```
Service: po.actioncon.com
Capabilities:
POST /orders
    In = application/vnd.com.actioncon.po+xml
GET /orders/{order-id}/status
    Out = text/plain
PUT /orders/{order-id}
    In = application/vnd.com.actioncon.po+xml
DELETE /orders/{order-id}
GET /orders/{order-id}
    Out = application/vnd.com.actioncon.po+xml
```

Example 10.3

The addition of a new resource or new supported method on a resource is a backwards-compatible change for a REST service.

As shown in Example 10.4, it may be important for service consumers to have a reasonable way of proceeding with their interaction if the service reports that the new method is not implemented.

```
Legal methods for actioncon.com service inventory:
* GET
* PUT
* DELETE
* POST
* SUBSCRIBE (consumers must fall back to periodic GET if service
reports "not implemented")
```

Example 10.4

New methods added to a service inventory's uniform contract need to provide a way for service consumers to "fall back" on a previously used method if they are to truly be backwards-compatible.

Changes to schemas and media types approach backwards compatibility in a different manner, in that they describe how information can be encoded for transport, and will often be used in both request and response messages. The focus for backwards compatibility is on whether a new message recipient can make sense of information sent by a legacy source. In other words, the new processor must continue to understand information produced by a legacy message *generator*.

An example of a change made to a schema for a message definition that is backwards-compatible is the addition of an optional element (as shown in bolded markup code in Example 10.5).

```
Media type = application/vnd.com.actioncon.po+xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="available" type="xsd:boolean"
    minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 10.5

In an XML Schema definition, the addition of an optional element is also considered backwards-compatible.

Here we are using a simplified version of the XML Schema definition for the Purchase Order service. The optional `available` element is added to the `LineItemType` complex type. This has no impact on existing generators because they are not required to provide this element in their messages. New processors must be designed to cope without the new information if they are to remain backwards-compatible.

Changing any of the existing elements in the previous example from required to optional (by adding the `minOccurs="0"` setting) would also be considered a backwards-compatible change. When we have control over how we choose to design the next version of a Web service contract, backwards compatibility is generally attainable. However, mandatory changes (such as those imposed by laws or regulations) can often force us to break backwards compatibility.

NOTE

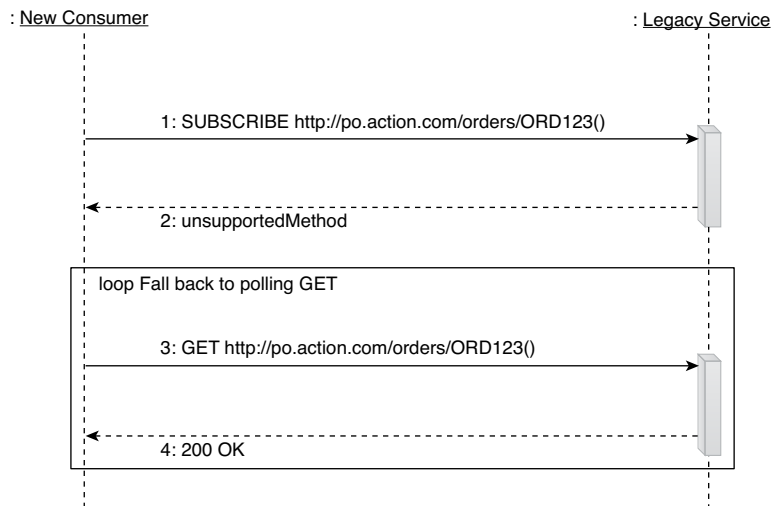
Both the Flexible and Loose versioning strategies explained at the end of this chapter support backwards compatibility.

Forwards Compatibility

When a service contract is designed in such a manner so that it can support a range of future consumer programs, it is considered to have an extent of *forwards compatibility*. This means that the contract can essentially accommodate how consumer programs will evolve over time.

Supporting forwards compatibility for Web service operations or uniform contract methods requires exception types to be present in the contract to allow service consumers to recover if they attempt to invoke a new and unsupported operation or method. For example, a “method not implemented” response enables the service consumer to detect that it is dealing with an incompatible service, thereby allowing it to handle this exception gracefully.

Redirection exception codes help REST services that implement a uniform contract change the resource identifiers in the contract when required. This is another way in which service contracts can allow legacy service consumers to continue using the service after contract changes have taken place (Example 10.6).



Example 10.6

A REST service ensures forwards compatibility by raising an exception whenever it does not understand a reusable contract or uniform contract method.

Forwards compatibility of schemas in REST services requires extension points to be present where new information can be added so that it will be safely ignored by legacy processors.

For example:

- Any validation that the processor does must not reject a document formatted according to the new schema.
- All existing information that the processor might need must remain present in future versions of the schema.
- Any new information added to the schema must be safe for legacy processors to ignore (if processors must understand the new information, then the change cannot be forwards compatible).
- The processor must ignore any information that it does not understand.

A common way to ensure validation does not reject future versions of the schema is to use wildcards in the earlier version. These provide extension points where new information can be added in future schema versions, as shown in Example 10.7.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:any namespace="##any" processContents="lax"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##any"/>
  </xsd:complexType>
</xsd:schema>
```

Example 10.7

To support forwards compatibility within a message definition generally requires the use of XML Schema wildcards.

In this example, the `xsd:any` and `xsd:anyAttribute` elements are added to allow for a range of unknown elements and data to be accepted by the service contract. In other words, the schema is being designed in advance to accommodate unforeseen changes in the future.

It is important to understand that building extension points into service contracts for forwards compatibility by no means eliminates the need to consider compatibility issues when making contract changes. New information can only be added to schemas

in a forwards-compatible manner if it is genuinely safe for processors to ignore. New operations are only able to be made forwards-compatible if a service consumer has an existing operation to fall back on when it finds the one it initially attempted to invoke is unsupported.

A service with a forwards-compatible contract will often not be able to process all message content. Its contract is simply designed to accept a broader range of data unknown at the time of its design.

NOTE

Forwards compatibility forms the basis of the Loose versioning strategy that is explained shortly.

Compatible Changes

When we make a change to a service contract that does not negatively affect existing consumers, then the change itself is considered a *compatible change*.

NOTE

In this book, the term “compatible change” refers to backwards compatibility by default. When used in reference to forwards compatibility, it is further qualified as a *forwards-compatible change*.

A simple example of a compatible change is when we set the `minOccurs` attribute of an element from “1” to “0,” effectively turning a required element into an optional one, as shown in Example 10.8.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType" />
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string" />
      <xsd:element name="productName" type="xsd:string"
        minOccurs="0" />
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```

    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

Example 10.8

The default value of the `minOccurs` attribute is "1". Therefore because this attribute was previously absent from the `productName` element declaration, it was considered a required element. Adding the `minOccurs="0"` setting turns it into an optional element, resulting in a compatible change. (Note that making this change to a message output from the service would be an incompatible change.)

This type of change will not impact existing consumer programs that are used to sending the element value to the Web service, nor will it affect future consumer programs that can be designed to optionally send that element.

Another example of a compatible change was provided earlier in Example 10.3, when we first added the optional `available` element declaration. Even though we extended the type with a whole new element, because it is optional it is considered a compatible change.

Here is a list of common compatible changes:

- Adding a new WSDL operation definition and associated message definitions
- Adding a new standard method to an existing REST resource
- Adding a set of new REST resources
- Changing the identifiers for a set of REST resources (including splitting and merging of services) using redirection response codes to facilitate migration of REST service consumers to the new identifiers
- Adding a new WSDL port type definition and associated operation definitions
- Adding new WSDL binding and service definitions
- Extending an existing uniform contract method in a way that can be safely ignored by REST services that can fall back on old service logic (for example, adding "If-None-Match" as a feature of the HTTP GET operation so that if the service ignores it, the consumer will still get the current and correct representation for the resource)
- Adding a new uniform contract method when an exception response exists for services that do not understand the method to use (and consumers can recover from this exception)

- Adding a new optional XML Schema element or attribute declaration to a message definition
- Reducing the constraint granularity of an XML Schema element or attribute of a message definition type used for input messages
- Adding a new XML Schema wildcard to a message definition type
- Adding a new optional WS-Policy assertion
- Adding a new WS-Policy alternative

Incompatible Changes

If after a change a contract is no longer compatible with consumers, then it is considered to have received an *incompatible change*. These are the types of changes that can break an existing contract and therefore impose the most challenges when it comes to versioning.

NOTE

The term “incompatible change” also indicates backwards compatibility by default. Incompatible changes that affect forwards compatibility will be qualified as “forwards-incompatible changes.”

Going back to our example, if we set an element’s `minOccurs` attribute from “0” to any number above zero, then we are introducing an incompatible change for input messages, as shown in Example 10.9:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="3"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 10.9

Incrementing the `minOccurs` attribute value of any established element declaration is automatically an incompatible change.

What was formerly an optional element is now required. This will certainly affect existing consumers that are not designed to comply with this new constraint, because adding a new required element introduces a mandatory constraint upon the contract.

Common incompatible changes include:

- Renaming an existing WSDL operation definition
- Removing an existing WSDL operation definition
- Changing the MEP of an existing WSDL operation definition
- Adding a fault message to an existing WSDL operation definition
- Adding a new required XML Schema element or attribute declaration to a message definition
- Increasing the constraint granularity of an XML Schema element or attribute declaration of a message definition
- Renaming an optional or required XML Schema element or attribute in a message definition
- Removing an optional or required XML Schema element or attribute or wildcard from a message definition
- Adding a new required WS-Policy assertion or expression
- Adding a new ignorable WS-Policy expression (most of the time)

Incompatible changes tend to cause most of the challenges with service contract versioning.

10.3 REST Service Compatibility Considerations

REST services within a given service inventory typically share a uniform contract for every resource, including uniform methods and media types. The same media types are used in both requests and responses, and new uniform contract facets are reused much more often than they are added to. This emphasis on service contract reuse within REST-compliant service inventories results in the need to highlight some special considerations, because changes to the uniform contract will automatically impact a range of service consumers because:

- The uniform contract methods are shared by all services.
- The uniform contract media types are shared by both services and service consumers.

As a result, both backwards compatibility and forwards compatibility considerations are almost equally important.

SOA PATTERNS

Service contracts that make use of the Schema Centralization [356] pattern without necessarily being REST-compliant will often need to impose a similarly rigid view of forwards compatibility and backwards compatibility.

Uniform contract methods codify the kinds of interactions that can occur between services and their consumers. For example, GET codifies “fetch some data,” while PUT codifies “store some data.”

Because the kinds of interactions that occur between REST services within the same service inventory tend to be relatively limited and stable, methods will usually change at a low rate compared to media types or resources. Compatibility issues usually pertain to a set of allowable methods that are only changed after careful case-by-case consideration.

An example of a compatible change to HTTP is the addition of `If-None-Match` headers to GET requests. If a service consumer knows the last version (or `etag`) of the resource that it fetched, it can make its GET request conditional. The `If-None-Match` header allows the consumer to state that the GET request should not be executed if the version of the resource is still the same as it was for the consumer’s last fetch. Instead, it will return the normal GET response, although it will do so in a non-optimal mode.

An example of an incompatible change to HTTP is the addition of a `Host` header used to support multihoming of Web servers. HTTP/1.0 did not require the name of the service to be included in request messages, but HTTP/1.1 does require this. If the special `Host` header is missing, HTTP/1.1 services must reject the request as being badly formed. However, HTTP/1.1 services are also required to be backwards-compatible, so if an HTTP/1.0 request comes into the REST service it will still be handled according to HTTP/1.0 rules.

Uniform contract media types further codify the kinds of information that can be exchanged between REST services and consumers. As previously stated, media types tend to change at a faster rate than HTTP methods in the uniform contract; however, media types still change more slowly than resources. Compatible change is more of a

live concern for the media types, and we can draw some more general rules about how to deal with them.

For example, if the generator of a message indicates to a processor of the message that it conforms to a particular media type, the processor generally does not need to know which version of the schema was used, nor does the processor need to have been built against the same version of the schema. The processor expects that all versions of the schema for a particular media type will be both forwards compatible and backwards-compatible with the type it was developed to support. Likewise, the generator expects that when it produces a message conformant with a particular schema version, that all processors of the message will understand it.

When incompatible changes are made to a schema, a new media type identifier is generally required to ensure that:

- The processor can decide how to parse a document based on the media type identifier
- Services and consumers are able to *negotiate* for a specific media type that will be understood by the processor when the message has been produced

Content negotiation is the ultimate fallback to ensure compatibility in REST-compliant service inventories. For a fetch interaction this often involves the consumer indicating to the service what media types it is able to support, and the service returning the most appropriate type that it supports. This mechanism allows for incompatible changes to be made to media types, as required.

NOTE

One way to better understand versioning issues that pertain to media types is to look at how they are used in HTML. An example of a compatible change to HTML that did not result in the need for a new media type was the addition of the `abbr` element to version 4.0 of the HTML language. This element allows new processors of HTML documents to support a mouse-over to expand abbreviations on a web page and to better support accessibility of the page. Legacy processors safely ignore the expansion, but will continue correctly showing the abbreviation itself.

An example of an incompatible change to HTML that did require a new media type was the conversion of HTML 4.0 to XML (resulting in version 1.0 of XHTML). The media type for the traditional SGML version remained `text/html`, while the XML version became `application/xhtml+xml`. This allowed content negotiation to occur between the two

types, and for processors to choose the correct parser and validation strategy based on which type was specified by the service.

Some incompatible changes have also been made to HTML without changing the media type. HTML 4.0 deprecated APPLET, BASEFONT, CENTER, DIR, FONT, ISINDEX, MENU, S, STRIKE, and U elements in favor of newer elements. These elements must continue to be understood but their use in HTML documents is being phased out. HTML 4.0 made LISTING, PLAINTEXT, and XMP obsolete. These elements should not be used in HTML 4.0 documents and no longer need to be understood.

Deprecating elements over a long period of time and eventually identifying them as obsolete once they are no longer used by existing services or consumers is a technique that can be used for REST media types to incrementally update a schema without having to change the media type.

10.4 Version Identifiers

One of the most fundamental design patterns related to Web service contract design is the Version Identification pattern. It essentially advocates that version numbers should be clearly expressed, not just at the contract level, but right down to the versions of the schemas that underlie the message definitions.

The first step to establishing an effective versioning strategy is to decide on a common means by which versions themselves are identified and represented within Web service contracts.

Versions are almost always communicated with version numbers. The most common format is a decimal, followed by a period and then another decimal, as shown here:

```
version="2.0"
```

Sometimes, you will see additional period plus decimal pairs that lead to more detailed version numbers like this:

```
version="2.0.1.1"
```

The typical meaning associated with these numbers is the measure or significance of the change. Incrementing the first decimal generally indicates a major version change (or upgrade) in the software, whereas decimals after the first period usually represent various levels of minor version changes.

From a compatibility perspective, we can associate additional meaning to these numbers. Specifically, the following convention has emerged in the industry:

- A minor version is expected to be backwards-compatible with other minor versions associated with a major version. For example, version 5.2 of a program should be fully backwards-compatible with versions 5.0 and 5.1.
- A major version is generally expected to break backwards compatibility with programs that belong to other major versions. This means that program version 5.0 is not expected to be backwards-compatible with version 4.0.

NOTE

A third “patch” version number is also sometimes used to express changes that are both forwards-compatible and backwards-compatible. Typically these versions are intended to clarify the schema only, or to fix problems with the schema that were discovered after it was deployed. For example, version 5.2.1 is expected to be fully compatible with version 5.2.0, but may be added for clarification purposes.

This convention of indicating compatibility through major and minor version numbers is referred to as the *compatibility guarantee*. Another approach, known as “amount of work,” uses version numbers to communicate the effort that has gone into the change. A minor version increase indicates a modest effort, and a major version increase predictably represents a lot of work.

These two conventions can be combined and often are. The result is often that version numbers continue to communicate compatibility as explained earlier, but they sometimes increment by several digits, depending on the amount of effort that went into each version.

There are various syntax options available to express version numbers. For example, you may have noticed that the declaration statement that begins an XML document can contain a number that expresses the version of the XML specification being used:

```
<?xml version="1.0"?>
```

That same `version` attribute can be used with the root `xsd:schema` element, as follows:

```
<xsd:schema version="2.0" ...>
```

You can further create a custom variation of this attribute by assigning it to any element you define (in which case you are not required to name the attribute “version”).

```
<LineItem version="2.0">
```

An alternative custom approach is to embed the major version number into a namespace or media type identifier, as shown here:

```
<LineItem xmlns="http://actioncon.com/schema/po/v2">
```

or

```
application/vnd.com.actioncon.po.v2+xml
```

Note that it has become a common convention to use date values in namespaces when versioning XML schemas, as follows:

```
<LineItem xmlns="http://actioncon.com/schema/po/2010/09">
```

In this case, it is the date of the change that acts as the major version identifier. To keep the expression of XML Schema definition versions in alignment with WSDL definition versions, we use version numbers instead of date values in upcoming examples. However, when working in an environment where XML Schema definitions are separately owned as part of an independent data architecture, it is not uncommon for schema versioning identifiers to be different from those used by WSDL definitions.

Regardless of which option you choose, it is important to consider the Canonical Versioning pattern that dictates that the expression of version information must be standardized across all service contracts within the boundary of a service inventory. In larger environments, this will often require a central authority that can guarantee the linearity, consistency, and description quality of version information. These types of conventions carry over into how service termination information is expressed, as further explored in Chapter 23 in *Web Service Contract Design and Versioning for SOA*.

SOA PATTERNS

Of course you may also be required to work with third-party schemas and WSDL definitions that may already have implemented their own versioning conventions. In this case, the extent to which the Canonical Versioning [327] pattern can be applied will be limited.

10.5 Versioning Strategies

There is no one versioning approach that is right for everyone. Because versioning represents a governance-related phase in the overall lifecycle of a service, it is a practice that is subject to the conventions, preferences, and requirements that are distinct to any enterprise.

Even though there is no de facto versioning technique for the WSDL, XML Schema, and WS-Policy content that comprises Web service contracts, a number of common and advocated versioning approaches have emerged, each with its own benefits and tradeoffs.

In this section, we single out the following three common strategies:

- *Strict* – Any compatible or incompatible changes result in a new version of the service contract. This approach does not support backwards or forwards compatibility.
- *Flexible* – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility but not forwards compatibility.
- *Loose* – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility and forwards compatibility.

These strategies are explained individually in the upcoming sections.

The Strict Strategy (New Change, New Contract)

The simplest approach to Web service contract versioning is to require that a new version of a contract be issued whenever any kind of change is made to any part of the contract.

This is commonly implemented by changing the target namespace value of a WSDL definition (and possibly the XML Schema definition) every time a compatible or incompatible change is made to the WSDL, XML Schema, or WS-Policy content related to the contract. Namespaces are used for version identification instead of a version attribute because changing the namespace value automatically forces a change in all consumer programs that need to access the new version of the schema that defines the message types.

This “super-strict” approach is not really that practical, but it is the safest and sometimes warranted when there are legal implications to Web service contract modifications, such as when contracts are published for certain interorganization data exchanges. Because both compatible and incompatible changes will result in a new contract version, this approach supports neither backwards nor forwards compatibility.

Pros and Cons

The benefit of this strategy is that you have full control over the evolution of the service contract, and because backwards and forwards compatibility are intentionally disregarded, you do not need to concern yourself with the impact of any change in particular (because all changes effectively break the contract).

On the downside, by forcing a new namespace upon the contract with each change, you are guaranteeing that all existing service consumers will no longer be compatible with any new version of the contract. Consumers will only be able to continue communicating with the Web service while the old contract remains available alongside the new version or until the consumers themselves are updated to conform to the new contract.

Therefore, this approach will increase the governance burden of individual services and will require careful transitioning strategies. Having two or more versions of the same service co-exist at the same time can become a common requirement for which the supporting service inventory infrastructure needs to be prepared.

The Flexible Strategy (Backwards Compatibility)

A common approach used to balance practical considerations with an attempt at minimizing the impact of changes to Web service contracts is to allow compatible changes to occur without forcing a new contract version, while not attempting to support forwards compatibility at all.

This means that any backwards-compatible change is considered safe in that it ends up extending or augmenting an established contract without affecting any of the service’s existing consumers. A common example of this is adding a new operation to a WSDL definition or adding an optional element declaration to a message’s schema definition.

As with the Strict strategy, any change that breaks the existing contract does result in a new contract version, usually implemented by changing the target namespace value of the WSDL definition and potentially also the XML Schema definition.

Pros and Cons

The primary advantage to this approach is that it can be used to accommodate a variety of changes while consistently retaining the contract's backwards compatibility. However, when compatible changes are made, these changes become permanent and cannot be reversed without introducing an incompatible change. Therefore, a governance process is required during which each proposed change is evaluated so that contracts do not become overly bloated or convoluted. This is an especially important consideration for agnostic services that are heavily reused.

The Loose Strategy (Backwards and Forwards Compatibility)

As with the previous two approaches, this strategy requires that incompatible changes result in a new service contract version. The difference here is in how service contracts are initially designed.

Instead of accommodating known data exchange requirements, special features from the WSDL, XML Schema, and WS-Policy languages are used to make parts of the contract intrinsically extensible so that they remain able to support a broad range of future, unknown data exchange requirements. For example:

- The `anyType` attribute value provided by the WSDL 2.0 language allows a message to consist of any valid XML document.
- XML Schema wildcards can be used to allow a range of unknown data to be passed in message definitions.
- Ignorable policy assertions can be defined to communicate service characteristics that can optionally be acknowledged by future consumers.

These and other features related to forwards compatibility are discussed in *Web Service Contract Design and Versioning for SOA*.

Pros and Cons

The fact that wildcards allow undefined content to be passed through Web service contracts provides a constant opportunity to further expand the range of acceptable message element and data content. On the other hand, the use of wildcards will naturally result in vague and overly coarse service contracts that place the burden of validation on the underlying service logic.

Strategy Summary

Provided in Table 10.1 is a broad summary of how the three strategies compare based on three fundamental characteristics.

	Strategy		
	<i>Strict</i>	<i>Flexible</i>	<i>Loose</i>
Strictness	High	Medium	Low
Governance Impact	High	Medium	High
Complexity	Low	Medium	High

Table 10.1

A general comparison of the three versioning strategies.

The three characteristics used in this table to form the basis of this comparison are as follows:

- *Strictness* – The rigidity of the contract versioning options. The Strict approach clearly is the most rigid in its versioning rules, while the Loose strategy provides the broadest range of versioning options due to its reliance on wildcards.
- *Governance Impact* – The amount of governance burden imposed by a strategy. Both Strict and Loose approaches increase governance impact but for different reasons. The Strict strategy requires the issuance of more new contract versions, which impacts surrounding consumers and infrastructure, while the Loose approach introduces the concept of unknown message sets that need to be separately accommodated through custom programming.
- *Complexity* – The overall complexity of the versioning process. Due to the use of wildcards and unknown message data, the Loose strategy has the highest complexity potential, while the straightforward rules that form the basis of the Strict approach make it the simplest option.

Throughout this comparison, the Flexible strategy provides an approach that represents a consistently average level of strictness, governance effort, and overall complexity.

10.6 REST Service Versioning Considerations

REST services that share the same uniform contract maintain separate versioned specifications for the following:

- The version number or specification of the resource identifier syntax (as per the “Request for Comments 6986 - Uniform Resource Identifier (URI): Generic Syntax” specification)
- The specification of the collection of legal methods, status codes, and other interaction protocol details (as per the “Request for Comments 2616 - Hypertext Transfer Protocol - HTTP/1.1” specification)
- Individual specifications for legal media types (for example, HTML 4.01 and the “Request for Comments 4287 - The Atom Syndication Format” specification)
- Individual specifications for service contracts that use the legal resource identifier syntax, methods, and media types

Each part of the uniform contract is specified and versioned independently of the others. Changing any one specification does not generally require another specification to be updated or versioned. Likewise, changing any of the uniform contract facet specifications does not require changes to individual service contracts, or changes to their version numbers.

This last point is in contradiction to some conventional versioning strategies. One might expect that if a schema used in a service contract changed, then the service contract would need to be modified. However, with REST services there is a tendency to maintain both forwards compatibility and backwards compatibility. If a REST service consumer sends a message that conforms to a newer schema, the service can process it as if it conformed to the older schema. If compatibility between these schemas has been maintained, then the service will function correctly. Likewise, if the service returns a message to the consumer that conforms to an old schema, the newer service consumer can still process the message correctly.

REST service contracts only need to directly consider the versioning of the uniform contract when media types used become deprecated, or when the schema advances so far that elements and attributes the service depends on are on their way to becoming obsolete. When this occurs, the service contract needs to be updated, and with it, the underlying service logic that processes the media types.

Part III



Appendices

Appendix A: Service-Oriented Principles Reference

Appendix B: REST Constraints Reference

Appendix C: SOA Design Patterns Reference

Appendix D: The Annotated SOA Manifesto

This page intentionally left blank